DEPSTAR

Devang Patel Institute of Advance Technology and Research

(A Constitute Institute of CHARUSAT)

Certificate

This is to certify that

Mr./~~Mrs.~~ Probin Bhagchandani

of 6CE1 *Class,*

ID. No. 22DCE006 *has satisfactorily completed*

his/ ~~her~~ term work in CE365 - Compeler Construction *for*

the ending in April 2024 /2025

Date : 1/4/25

MC

Sign. of Faculty

Dgarg

Head of Department

Faculty of Technology and Engineering

Computer Engineering / Computer Science and Engineering

Date : 16 / 12 / 2024

Practical List

Academic Year	:	2024-2025	Semester	:	6
Course code	:	CE365 CSE313	Course name	:	Compiler Construction Design of Language Processor

Sr. No.	Aim	Hr	CO
1.	**Practical Definition** String Validation Against Regular Expression **Objective** To implement a program that validates a user-input string against the regular expression a*bb. The program should determine whether the input string is valid or invalid based on the defined pattern. **Language Constraint** The program must be implemented in C language. **Input Requirements** • Accept a character string from the user. • Ensure the input is terminated with a newline character. **Expected output** • If the input string matches the pattern a*bb, the program should output: "Valid String". • If the input string does not match the pattern, the program should output: "Invalid String". **Sample input output** Input: aaabb → Output: Valid string Input: Abab → Output: Invalid string **Testcases** ^ bbbb aaa baaabb aaabbb baaabb aaaab abbabb abb aaaaabb	2	1

2.	**Practical Definition** String Validation Using Finite Automata **Objective** To implement a program that validates a given string against rules defined in terms of finite automata. **Language Constraint** The program can be implemented in any programming language **Input requirement** • Accept rules in the form of finite automata (e.g., states, transitions, start state, accept states) as input. • Accept a string to be validated against the provided finite automata rules. **Expected output** • If the string adheres to the rules of the finite automata, the program should output: "Valid String". • If the string does not adhere to the rules, the program should output: "Invalid String". **Sample input output** Input: Number of input symbols : 2 Input symbols : a b Enter number of states : 4 Initial state : 1 Number of accepting states : 1 Accepting states : 2 Transition table : 1 to a -> 2 1 to b -> 3 2 to a -> 1 2 to b -> 4 3 to a -> 4 3 to b -> 1 4 to a -> 3 4 to b -> 2 Input string : abbabab Output: Valid string **Testcases** • String over 0 and 1 where every 0 immediately followed by 11 • string over a b c, starts and end with same letter. • String over lower-case alphabet and digits, starts with alphabet only.	2	1

3.	**Practical Definition** Implementation of a Lexical Analyzer for C Language Compiler **Objective** To design and implement a lexical analyser, the first phase of a compiler, for the C programming language. The lexical analyser should perform the following tasks: (1) tokenizing the input string (2) removing comments (3) removing white spaces (4) entering identifiers into the symbol table (5) generating lexical errors. **Language Constraint** The program can be implemented in any programming language **Input requirement** • Accept a C source code file. • The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces. **Expected output** • Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator. • Symbol table with all identified identifiers stored. • Detection and reporting of lexical errors • Modified source code **Sample input output**	4	1

Input	Output	

int main() {
int a = 5 , 7H;
// assign value
char b = 'x';
/* return
value */
return a + b;
}
	TOKENS
	Keyword: int
	Identifier: main
	Punctuation: (
	Punctuation:)
	Punctuation: {
	Keyword: int
	Identifier: a
	Operator: =
	Constant: 5
	Punctuation : ,
	Punctuation: ;
	Keyword: char
	Identifier: b
	Operator: =
	String: 'x'
	Punctuation: ;
	Keyword: return
	Identifier: a
	Operator: +
	Identifier: b
	Punctuation: ;
	Punctuation: }
		LEXICAL ERRORS
		7H invalid lexeme
		
		SYMBOL TABLE ENTRIES
		1) a
		2) b

Testcases

/* salary calculation*/ void main( )	// user defined data type struct student

```
{
long int bs , da , hra , gs;
//take basic salary as input
scanf("%ld",&bs);
//calculate allowances
da=bs*.40;
hra=bs*.20;
gs=bs+da+hra;
// display salary slip
printf("\n\nbs  : %ld",bs);
printf("\nda  : %ld",da);
printf("\nhra : %ld",hra);
printf("\ngs  : %ld",gs);
}
```

```
{
int id;
float cgpa;
}
void main( )
{
student s;
s.id = 10;
s.cgpa = 8.7;
}
```

```
//function prototype
void add ( int , int );
void main( )
{
int a , b;
a = 10;
b = 20;
// function call
add ( a , b );
}
void add ( int x , int y )
{
return x + y;
}
```

4.	**Practical Definition** String validation using Lax tool **Objective - 1** Write a program to identify and extract all numbers from input string and display them one by one in new line. **Language Constraint** Lex (Lexical analyser generator) **Input requirement** • Accept a character string, mix of text and numbers, from the user. • Ensure the input is terminated with a newline character. **Expected output** The program should print out each number found in the input, each on a new line. **Sample input output** Input: a1b22c3 Output: 1 22 3 **Testcases** power operation -> 12 ** 3 = 1728 You multiply 804569 with 1 then will be : **Objective - 2** Write a program to replace the word "charusat" with “university” in the input text. **Language Constraint** Lex (Lexical analyser generator) **Input requirement** • Accept a character string from the user where the word "charusat" may appear multiple times. • Ensure the input is terminated with a newline character. **Expected output** The program should print the input text with all occurrences of "charusat" replaced by “university”. **Sample input output** Input: This is charusat. Output: This is university. **Testcases** Charusat is in Anand district. / I am doing my BTech from CHARSAT. Charusat , What is charusat? / Every where it is charusat , charusat and only charusat.	4	1

Objective – 3
Write a program to count number of characters, word and lines from the input file.

Language Constraint
Lex (Lexical analyser generator)

Input requirement
Read contain from a text file containing multiple word and lines.

Expected output
The program should print total number of characters (including spaces), words (separated by white spaces), lines (end with new line symbol).

Sample input output

Input	Output
The 45 is odd number.	Characters : 22 Words : 5 Line : 1

Testcases

I want to calculate a number. The number of characters, words and lines. All know that \n is ending character of line.
45 + 89 =40

Objective – 4
Write a program which validate the password as per given rules.
- length can be 9 to 15 characters
- includes lower case letter, upper case letter, digit, symbols (*, ; # $ @)
- minimum count for each category must be one

Language Constraint
Lex (Lexical analyser generator)

Input requirement
- Accept a character string from the user which is mix of letters, numbers and symbols.
- Ensure the input is terminated with a newline character.

Expected output
- If the password meets the given rules, the program should print "Valid password".
- If the password does not meet the rules, the program should print "Invalid password".

Sample input output

Input	Output
a@1T	Invalid password

Testcase

	aB1@	aaBB11,#cdefg2345	CHARUSAT		
	Charusat	CHArusat123	Cspit-2024		
	Charusat@2024	Charu$at@20#24	charu*sAT;22		

5. **Practical Definition**

Implementation of a Lexical Analyzer for C Language Compiler

Objective

To design and implement a lexical analyser to perform 1st, 2nd, 3rd, and 5th task as per the list given in practical 2.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

- Accept a C source code file.
- The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output

- Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.
- Detection and reporting of lexical errors

Sample input output

Input	Output

int main() {
int a = 5 , 7H;
// assign value
char b = 'x';
/* return
value */
return a + b;
}
	TOKENS
	Keyword: int
	Identifier: main
	Punctuation: (
	Punctuation:)
	Punctuation: {
	Keyword: int
	Identifier: a
	Operator: =
	Constant: 5
	Punctuation : ,
	Punctuation: ;
	Keyword: char
	Identifier: b
	Operator: =
	String: 'x'
	Punctuation: ;
	Keyword: return
	Identifier: a
	Operator: +
	Identifier: b
	Punctuation: ;
	Punctuation: }

Testcases

/* salary calculation*/
void main( )
{
long int bs , da , hra , gs;
//take basic salary as input
scanf("%ld",&bs);
//calculate allowances
	// user defined data type
	struct student
	{
	int id;
	float cgpa;
	}
	void main( )

2 | 1

	da=bs*.40;
	hra=bs*.20;
	gs=bs+da+hra;
	// display salary slip
	printf("\n\nbs : %ld",bs);
	printf("\nda : %ld",da);
	printf("\nhra : %ld",hra);
	printf("\ngs : %ld",gs);
	}
		{
		student s;
		s.id = 10;
		s.cgpa = 8.7;
		}			

	//function prototype
	void add (int , int);
	void main( )
	{
	int a , b;
	a = 10;
	b = 20;
	// function call
	add (a , b);
	}
	void add (int x , int y)
	{
	return x + y;
	}				

6.	**Practical definition** String validation using Recursive Descent Parsing (RDP) **Objective** Implement a Recursive Descent Parser (RDP) to validate an input string against the given grammar. $S \rightarrow (L) \mid a$ $L \rightarrow S L'$ $L' \rightarrow , S L' \mid \epsilon$ **Language constraint** The program can be implemented in any programming language **Input Requirement** A string that needs to be validated against the grammar **Expected output** • If the input string is valid according to the grammar, the program should print "Valid string". • If the input string is invalid according to the grammar, the program should print "Invalid string". **Sample input output** Input: (a) → Output: Valid string **Testcases** a, (a), (a,a), (a,(a,a),a), (a,a),(a,a) a), (a, a,a, a,, (a,a),a	2	2

7.	**Program definition** Computing First and Follow Sets for a Context-Free Grammar (CFG) **Objective** Develop a program computes the First and Follow sets for all non-terminal symbols in for the below given grammar. S → A B C | D A → a | ε B → b | ε C → (S) | c D → A C **Language Constraint** The program can be implemented in any programming language **Input requirement** No input **Expected output** First(S) = {a, b, (, c} First(A) = {a, ε} First(B) = {b, ε} First(C) = {(, c} First(D) = {a, (} Follow(S) = {), $} Follow(A) = {b, (,), $} Follow(B) = {c,), $} Follow(C) = {), $} Follow(D) = {), $}	2	2

8.	**Program definition** Predictive Parsing Table Construction and LL(1) Grammar Validation **Objective** Develop a program to construct a predictive parsing table for the given grammar. The program should analyse the generated parsing table to determine whether the grammar is LL(1) or not. If the grammar is LL(1), the program should also validate an input string against the given grammar. **Language Constraint** The program can be implemented in any programming language **Input requirement** • First() and Follow() generated by practical 7 • A string that needs to be validated against the grammar **Expected output** • A predictive parsing table generated for the given grammar. • A message indicating whether the grammar is LL(1) or not. • If the input string is valid according to the grammar, the program should print "Valid string". • If the input string is invalid according to the grammar, the program should print "Invalid string". **Testcases** abc, ac, (abc), c, (ac) a, ( ), (ab), abcabc, b	3	2

9.	**Program definition** String parsing using YACC **Objective** Develop a YACC program to validate input strings based on the given grammar. The program should parse the string using the grammar rules and determine whether the string is valid or invalid. S → i E t S S' | a S' → e S | ε E → b **Language Constraint** YACC (Syntex Analyser generator) **Input requirement** An input string to validate based on the provided grammar. **Expected output** • If the input string is valid according to the grammar, the program should print "Valid string". • If the input string is invalid according to the grammar, the program should print "Invalid string". **Sample input output** Input: i b t — Output: Invalid string **Testcases** ibtai, ibtaea, a, ibtibta, ibtaeibta ibti, ibtaa, iea, ibtb, ibtibt	3	2

10.	**Program definition** Evaluating Arithmetic Expression with Bottom-Up Approach Using SDD **Objective** Develop a program to evaluate arithmetic expressions containing operators using a bottom-up parsing approach and below given Syntax-Directed Definitions (SDD) for semantic evaluation. The program will compute the result of the expression by building a parse tree using and will incorporate semantic rules to evaluate sub-expressions during parsing.	2	4

L → E n	Print (E.val)
E → E + T	E.Val = E.val + T.val
E → E - T	E.Val = E.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → T / F	T.val = T.val / F.val
T → F	T.val = F.val
F → G ^ F	F.val = G.val ^ F.val
F → G	F.val = G.val
G → (E)	G.val = E.val
G → digit	G.val = digit.lexval

Language Constraint
An input string

Input requirement
An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75)
- Operators: +, -, *, /, ^ for addition, subtraction, multiplication, division, and exponentiation
- Parentheses: (and) for grouping sub-expressions

Expected output

- The evaluated result of the arithmetic expression.
- If the input expression is invalid, the program should display "Invalid expression".

Sample input output

Input	Output
(3 + 5) * 2 ^ 3	64

Testcases

(3 + 5) * 2	3 + 5 * 2	3 + 5 * 2 ^ 2	3 + (5 * 2)	3 + 5 ^ 2 * 2
3 * (5 + 2)	(3 + 5) ^ 2	3 ^ 2 ^ 3	3 ^ 2 + 5 * 2	3 + ^ 5
(3 + 5 * 2	(3 + 5 * 2 ^ 2 - 8) / 4 ^ 2 + 6			

11. **Program definition** — 2 — 4

Generate Intermediate Code Using Quadruple Table

Objective

Develop a program that break down the input string according to the grammar and produce a sequence of quadruples representing the intermediate code for the expression.

E → E + T | E – T | T
T → T * F | T / F | F
F → (E) | digit

Language Constraint

The program can be implemented in any programming language

Input requirement

An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75)
- Operators: +, -, *, / for addition, subtraction, multiplication, division, and exponentiation
- Parentheses: (and) for grouping sub-expressions

Expected output

A quadruple table representing the intermediate code for the given expression

Sample input output

Input	Output			
9 + 42 * 8	Operator	Operand 1	Operand 2	Result
	*	42	8	t1
	+	9	t1	t2

Testcases

5 + 6 – 3	7 – (8 * 2)	(9 – 3) + (5 * 4 / 2)
(3 + 5 * 2 - 8) / 4 – 2 + 6		86 / 2 / 3

<table>
<tr>
<td>12.</td>
<td>
Program definition
Code Optimization Using Constant Folding

Objective
Develop a program that identifies constant expressions at compile-time and replaces them with their evaluated results to enhance execution efficiency.

Language Constraint
The program can be implemented in any programming language

Input requirement
An arithmetic expression in the form of a string that can contain
• Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75) or variables (e.g. a , abc , cgpa)
• Operators: +, -, *, / for addition, subtraction, multiplication, division, and exponentiation

Expected output
Display the optimized expression after applying constant folding

Sample input output
<table>
<tr><td>Input</td><td>Output</td></tr>
<tr><td>5 + x - 3 * 2</td><td>5 + x - 6</td></tr>
</table>

Testcases
<table>
<tr><td>2 + 3 * 4 - 1</td><td>x + (3 * 5) - 2</td><td>(22 / 7) * r * r</td></tr>
</table>
</td>
<td>2</td>
<td>5</td>
</tr>
</table>

PRACTICAL: 1

AIM: String Validation Against Regular Expression

Objective

To implement a program that validates a user-input string against the regular expression a*bb. The program should determine whether the input string is valid or invalid based on the defined pattern.

Language Constraint

The program must be implemented in C language.

Input Requirements

- Accept a character string from the user.
- Ensure the input is terminated with a newline character.

Expected output

- If the input string matches the pattern a*bb, the program should output: "Valid String".
- If the input string does not match the pattern, the program should output: "Invalid String".

Sample input output

Input	Output
aaabb	Valid string
Abab	Invalid string

Testcases

^	bbbb	aaa	baaabb	aaabbb
baaabb	aaaab	abbabb	abb	aaaaabb

CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
bool isValidString(const char *str) {
   int length = strlen(str);
   if (length < 2) {
      return false;
   }
   if (str[length - 1] != 'b' || str[length - 2] != 'b') {
```

```
        return false;
    }
    for (int i = 0; i < length - 2; i++) {
        if (str[i] != 'a') {
            return false;
        }
    }
    return true;
}
int main() {
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);
    if (isValidString(input)) {
        printf("Valid String\n");
    } else {
        printf("Invalid String\n");
    }
    return 0;
}
```

OUTPUT:

```
Enter a string: aaabb
Valid String

Process returned 0 (0x0)   execution time : 3.209 s
Press any key to continue.
```

PRACTICAL: 2

AIM: String Validation Using Finite Automata

Objective

To implement a program that validates a given string against rules defined in terms of finite automata.

Language Constraint

The program can be implemented in any programming language

Input requirement

- Accept rules in the form of finite automata (e.g., states, transitions, start state, accept states) as input.
- Accept a string to be validated against the provided finite automata rules.

Expected output

- If the string adheres to the rules of the finite automata, the program should output: "Valid String".
- If the string does not adhere to the rules, the program should output: "Invalid String".

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_STATES 100
#define MAX_SYMBOLS 10

// Structure to define the finite automaton
typedef struct {
    int num_states;
    int num_symbols;
    char symbols[MAX_SYMBOLS];
    int initial_state;
    int accepting_states[MAX_STATES];
    int transition_table[MAX_STATES][MAX_SYMBOLS];
} FiniteAutomaton;

// Function to find the index of a symbol
int find_symbol_index(FiniteAutomaton *fa, char symbol) {
    for (int i = 0; i < fa->num_symbols; i++) {
        if (fa->symbols[i] == symbol) {
            return i;
        }
    }
    return -1; // Symbol not found
```

```
}

// Function to simulate the finite automaton and validate the string
int validate_string(FiniteAutomaton *fa, char *input_string) {
    int current_state = fa->initial_state;

    // Process each symbol in the input string
    for (int i = 0; i < strlen(input_string); i++) {
        char symbol = input_string[i];
        int symbol_index = find_symbol_index(fa, symbol);

        if (symbol_index == -1) {
            return 0; // Invalid symbol found in the input string
        }

        // Move to the next state based on the transition table
        current_state = fa->transition_table[current_state][symbol_index];
    }

    // Check if the final state is an accepting state
    for (int i = 0; i < fa->num_states; i++) {
        if (fa->accepting_states[i] == current_state) {
            return 1; // Valid string
        }
    }
    return 0; // Invalid string
}

int main() {
    FiniteAutomaton fa;
    char input_string[100];

    // Read the number of input symbols
    printf("Number of input symbols: ");
    scanf("%d", &fa.num_symbols);

    // Read the input symbols
    printf("Input symbols: ");
    for (int i = 0; i < fa.num_symbols; i++) {
        scanf(" %c", &fa.symbols[i]);
    }

    // Read the number of states
    printf("Enter number of states: ");
    scanf("%d", &fa.num_states);
```

```
// Read the initial state
printf("Initial state: ");
scanf("%d", &fa.initial_state);

// Read the accepting states
int num_accepting_states;
printf("Number of accepting states: ");
scanf("%d", &num_accepting_states);
printf("Accepting states: ");
for (int i = 0; i < num_accepting_states; i++) {
    scanf("%d", &fa.accepting_states[i]);
}

// Initialize transition table to -1
for (int i = 0; i < fa.num_states; i++) {
    for (int j = 0; j < fa.num_symbols; j++) {
        fa.transition_table[i][j] = -1;
    }
}

// Read the transition table
printf("Transition table (format: state input_symbol next_state state):\n");
for (int i = 0; i < fa.num_states * fa.num_symbols; i++) {
    int state, next_state;
    char symbol;
    if (scanf("%d %c %d", &state, &symbol, &next_state) != 3) {
        break; // Break the loop if input is exhausted
    }
    int symbol_index = find_symbol_index(&fa, symbol);
    if (symbol_index != -1) {
        fa.transition_table[state][symbol_index] = next_state;
    }
}

// Read the input string to be validated
printf("Input string: ");
scanf("%s", input_string);

// Validate the string and print the result
if (validate_string(&fa, input_string)) {
    printf("Valid String\n");
} else {
    printf("Invalid String\n");
}
```

```
    return 0;
}
```

OUTPUT:

```
Number of input symbols: 2
Input symbols: a b
Enter number of states: 4
Initial state: 1
Number of accepting states: 1
Accepting states: 2
Transition table (format: state input_symbol next_state state):
1 a 2
1 b 3
2 a 1
2 b 4
3 a 4
3 b 1
4 a 3
4 b 2
Input string: abbabab
Valid String

Process returned 0 (0x0)   execution time : 53.166 s
Press any key to continue.
```

PRACTICAL: 3

AIM: Implementation of a Lexical Analyzer for C Language Compiler

Objective

To design and implement a lexical analyser, the first phase of a compiler, for the C programming language. The lexical analyser should perform the following tasks: (1) tokenizing the input string (2) removing comments (3) removing white spaces (4) entering identifiers into the symbol table (5) generating lexical errors.

Language Constraint

The program can be implemented in any programming language

Input requirement

- Accept a C source code file.
- The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output

- Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.
- Symbol table with all identified identifiers stored.
- Detection and reporting of lexical errors
- Modified source code

CODE:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define MAX_TOKEN_LENGTH 100
#define MAX_SYMBOL_TABLE_SIZE 100

typedef enum {
   KEYWORD, IDENTIFIER, CONSTANT, STRING, PUNCTUATION, OPERATOR, ERROR
} TokenType;

const char *keywords[] = {"int", "char", "return", "if", "else", "while", "for", "do", "break", "continue",
"void", "static", "struct", "union", "enum", "typedef", "sizeof", "switch", "case", "default", "const",
"volatile", "register", "extern", "auto", "signed", "unsigned", "short", "long", "float", "double"};
```

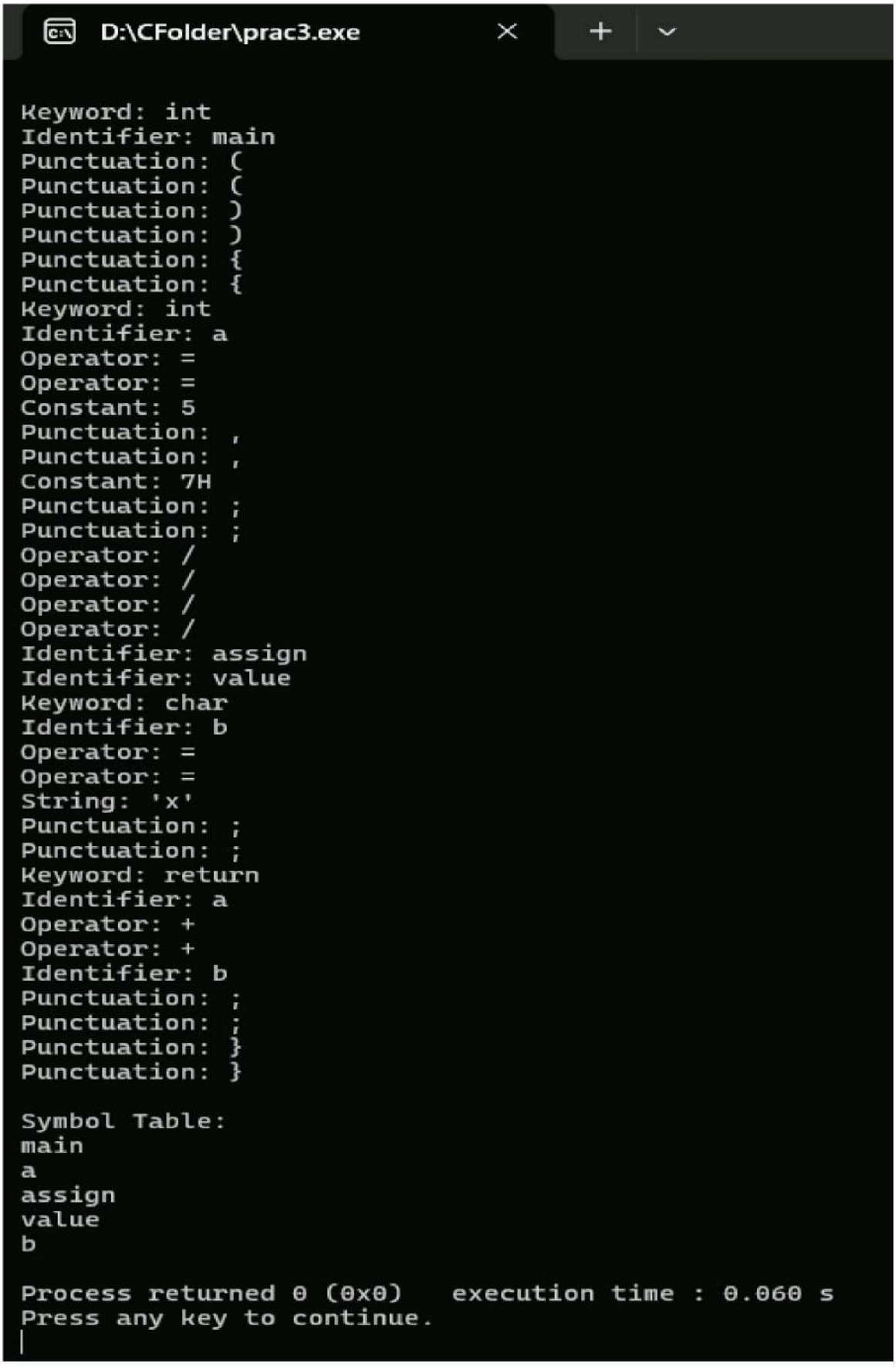
```
const char *operators[] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "!",
"&", "|", "^", "~", "<<", ">>", "++", "--"};
const char *punctuations[] = {"(", ")", "{", "}", "[", "]", ";", ",", ".", ":", "?"};

typedef struct {
    char name[MAX_TOKEN_LENGTH];
    TokenType type;
} Token;

typedef struct {
    char name[MAX_TOKEN_LENGTH];
} Symbol;

Symbol symbolTable[MAX_SYMBOL_TABLE_SIZE];
int symbolTableSize = 0;

void addSymbol(char *name) {
    for (int i = 0; i < symbolTableSize; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return;
        }
    }
    strcpy(symbolTable[symbolTableSize++].name, name);
}

TokenType getTokenType(char *token) {
    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
        if (strcmp(token, keywords[i]) == 0) {
            return KEYWORD;
        }
    }
    for (int i = 0; i < sizeof(operators) / sizeof(operators[0]); i++) {
        if (strcmp(token, operators[i]) == 0) {
            return OPERATOR;
        }
    }
    for (int i = 0; i < sizeof(punctuations) / sizeof(punctuations[0]); i++) {
        if (strcmp(token, punctuations[i]) == 0) {
            return PUNCTUATION;
        }
    }
    if (isdigit(token[0])) {
        return CONSTANT;
    }
    if (token[0] == '\" || token[0] == '\"') {
```

```
        return STRING;
    }
    if (isalpha(token[0]) || token[0] == '_') {
        addSymbol(token);
        return IDENTIFIER;
    }
    return ERROR;
}

void printToken(Token token) {
    const char *typeNames[] = {"Keyword", "Identifier", "Constant", "String", "Punctuation",
"Operator", "Error"};
    printf("%s: %s\n", typeNames[token.type], token.name);
}

void tokenize(char *source) {
    char token[MAX_TOKEN_LENGTH];
    int tokenIndex = 0;
    int inComment = 0;
    int inString = 0;

    for (int i = 0; source[i] != '\0'; i++) {
        char c = source[i];

        if (inComment) {
            if (c == '*' && source[i + 1] == '/') {
                inComment = 0;
                i++;
            }
            continue;
        }

        if (inString) {
            token[tokenIndex++] = c;
            if (c == '\"' || c == '\'') {
                token[tokenIndex] = '\0';
                Token t = { .type = STRING };
                strcpy(t.name, token);
                printToken(t);
                tokenIndex = 0;
                inString = 0;
            }
            continue;
        }
```

```
    if (c == '/' && source[i + 1] == '*') {
        inComment = 1;
        i++;
        continue;
    }

    if (isspace(c)) {
        if (tokenIndex > 0) {
            token[tokenIndex] = '\0';
            Token t = { .type = getTokenType(token) };
            strcpy(t.name, token);
            printToken(t);
            tokenIndex = 0;
        }
        continue;
    }

    if (c == '"' || c == '\'') {
        if (tokenIndex > 0) {
            token[tokenIndex] = '\0';
            Token t = { .type = getTokenType(token) };
            strcpy(t.name, token);
            printToken(t);
            tokenIndex = 0;
        }
        inString = 1;
        token[tokenIndex++] = c;
        continue;
    }

    for (int j = 0; j < sizeof(punctuations) / sizeof(punctuations[0]); j++) {
        if (c == punctuations[j][0]) {
            if (tokenIndex > 0) {
                token[tokenIndex] = '\0';
                Token t = { .type = getTokenType(token) };
                strcpy(t.name, token);
                printToken(t);
                tokenIndex = 0;
            }
            Token t = { .type = PUNCTUATION };
            t.name[0] = c;
            t.name[1] = '\0';
            printToken(t);
            break;
        }
```

```
        }

        for (int j = 0; j < sizeof(operators) / sizeof(operators[0]); j++) {
            if (c == operators[j][0]) {
                if (tokenIndex > 0) {
                    token[tokenIndex] = '\0';
                    Token t = { .type = getTokenType(token) };
                    strcpy(t.name, token);
                    printToken(t);
                    tokenIndex = 0;
                }
                Token t = { .type = OPERATOR };
                t.name[0] = c;
                t.name[1] = '\0';
                printToken(t);
                break;
            }
        }

        token[tokenIndex++] = c;
    }

    if (tokenIndex > 0) {
        token[tokenIndex] = '\0';
        Token t = { .type = getTokenType(token) };
        strcpy(t.name, token);
        printToken(t);
    }
}

int main() {
    printf("\n");
    char source[] = "int main() {\nint a = 5 , 7H;\n// assign value\nchar b = 'x';\n/* return\nvalue
*/\nreturn a + b;\n}";

    tokenize(source);

    printf("\nSymbol Table:\n");
    for (int i = 0; i < symbolTableSize; i++) {
        printf("%s\n", symbolTable[i].name);
    }

    return 0;
}
```

OUTPUT:


D:\CFolder\prac3.exe
Keyword: int
Identifier: main
Punctuation: (
Punctuation: (
Punctuation:)
Punctuation:)
Punctuation: {
Punctuation: {
Keyword: int
Identifier: a
Operator: =
Operator: =
Constant: 5
Punctuation: ,
Punctuation: ,
Constant: 7H
Punctuation: ;
Punctuation: ;
Operator: /
Operator: /
Operator: /
Operator: /
Identifier: assign
Identifier: value
Keyword: char
Identifier: b
Operator: =
Operator: =
String: 'x'
Punctuation: ;
Punctuation: ;
Keyword: return
Identifier: a
Operator: +
Operator: +
Identifier: b
Punctuation: ;
Punctuation: ;
Punctuation: }
Punctuation: }
Symbol Table:
main
a
assign
value
b
Process returned 0 (0x0) execution time : 0.060 s
Press any key to continue.

PRACTICAL: 4

AIM: String validation using Lax tool

Objective - 1

Write a program to identify and extract all numbers from input string and display them one by one in new line.
Language Constraint
Lex (Lexical analyser generator)
Input requirement
• Accept a character string, mix of text and numbers, from the user.
• Ensure the input is terminated with a newline character.
Expected output
The program should print out each number found in the input, each on a new line.

CODE:

```
%{

#include <stdio.h>
%}
%%
[0-9]+ {
   printf("%s\n", yytext);
}

[^0-9\n]+ {
}

\n {
}
%%

int main() {
   printf("Enter a string (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):\n");
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

OUTPUT:

```
Test Case 1:
Enter a string (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
a1b22c3
1
22
3

Test Case 2:
Enter a string (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
power operation -> 12 ** 3 = 1728
12
3
1728

Test Case 3:
Enter a string (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
You multiply 804369 with 1 then will be :
804369
1
```

Objective - 2

Write a program to replace the word "charusat" with “university” in the input text.
Language Constraint
Lex (Lexical analyser generator)
Input requirement
• Accept a character string from the user where the word "charusat" may appear multiple times.
• Ensure the input is terminated with a newline character.
Expected output
The program should print the input text with all occurrences of "charusat" replaced by “university”.

CODE :

```
%{
#include <stdio.h>
#include <string.h>
%}

%%
[cC][hH][aA][rR][uU][sS][aA][tT] {
   printf("university");
}

. {
   printf("%s", yytext);
}

\n {
```

```
    printf("\n");
}
%%

int main() {
    printf("Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):\n");
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

OUTPUT:

```
Test Case 1:
Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
This is charusat.
This is university.

Test Case 2:
Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
Charusat is in Anand district
university is in Anand district

Test Case 3:
Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
I am doing my BTech from CHARUSAT
I am doing my BTech from university

Test Case 4:
Enter text (press Ctrl+D on Unix/Linux or Ctrl+Z on Windows to end input):
Every where it is charusat , charusat and only charusat
Every where it is university , university and only university
```

Objective – 3

Write a program to count number of characters, word and lines from the input file.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

Read contain from a text file containing multiple word and lines.

Expected output

The program should print total number of characters (including spaces), words (separated by white spaces), lines (end with new line symbol).

CODE :

```
%{
#include <stdio.h>

int char_count = 0;
int word_count = 0;
int line_count = 0;
int in_word = 0;
%}

%%

[^ \t\n]+ {
   word_count++;
   char_count += yyleng;  /* Add length of the word to character count */
}

[ \t]+ {
   char_count += yyleng;
}

\n {
   line_count++;
   char_count++;
}

%%
int main(int argc, char* argv[]) {
   if (argc > 1) {
      FILE* file = fopen(argv[1], "r");
      if (!file) {
         printf("Could not open file %s\n", argv[1]);
         return 1;
      }
      yyin = file;
   } else {
      printf("Please provide an input file.\n");
      printf("Usage: %s <filename>\n", argv[0]);
      return 1;
   }

   yylex();

   printf("Characters : %d\n", char_count);
   printf("Words : %d\n", word_count);
   printf("Line : %d\n", line_count);

   if (argc > 1) {
      fclose(yyin);
   }
```

```
    return 0;
}
int yywrap() {
    return 1;
}
```

OUTPUT:

```
Characters : 22
Words : 5
Line : 1
```

```
Characters : 74
Words : 13
Line : 1
```

Objective – 4

Write a program which validate the password as per given rules.

➢

length can be 9 to 15 characters

➢

includes lower case letter, upper case letter, digit, symbols (*, ; # $ @)

➢

minimum count for each category must be one

Language Constraint

Lex (Lexical analyser generator)

Input requirement

•

Accept a character string from the user which is mix of letters, numbers and symbols.

•

Ensure the input is terminated with a newline character.

Expected output

•

If the password meets the given rules, the program should print "Valid password".

•

If the password does not meet the rules, the program should print "Invalid password".

CODE :

```
%{
#include <stdio.h>
#include <string.h>

int length = 0;
int has_lower = 0;
int has_upper = 0;
int has_digit = 0;
int has_symbol = 0;
```

```
void reset_checks() {
   length = 0;
   has_lower = 0;
   has_upper = 0;
   has_digit = 0;
   has_symbol = 0;
}

void validate_password() {
   if (length >= 9 && length <= 15 &&
      has_lower && has_upper && has_digit && has_symbol) {
      printf("Valid password\n");
   } else {
      printf("Invalid password\n");
   }
   reset_checks();
}
%}

%%

^[^\n]+ {
   length = yyleng;
   for (int i = 0; i < yyleng; i++) {
      if (yytext[i] >= 'a' && yytext[i] <= 'z') {
         has_lower = 1;
      } else if (yytext[i] >= 'A' && yytext[i] <= 'Z') {
         has_upper = 1;
      } else if (yytext[i] >= '0' && yytext[i] <= '9') {
         has_digit = 1;
      } else if (yytext[i] == '*' || yytext[i] == '+' ||
            yytext[i] == '#' || yytext[i] == '$' ||
            yytext[i] == '@') {
         has_symbol = 1;
      }
   }
   validate_password();
}

\n  { }

%%
int main() {
   printf("Enter a password to validate");
   yylex();
   return 0;
}
int yywrap() {
   return 1;
}
```

OUTPUT:

```
Test Case 1:
Enter a password to validate:
a@1T
Invalid password

Test Case 2:
Enter a password to validate:
a$1@
Invalid password

Test Case 3:
Enter a password to validate:
aaRR11#csfz345
Valid password

Test Case 4:
Enter a password to validate:
CHarusat123
Invalid password

Test Case 5:
Enter a password to validate:
Cgpit-2024
Invalid password

Test Case 6:
Enter a password to validate:
Charusat@2024
Valid password

Test Case 7:
Enter a password to validate:
Charusat@20$24
Valid password
```

PRACTICAL: 5

AIM:

Practical Definition

Implementation of a Lexical Analyzer for C Language Compiler

Objective

To design and implement a lexical analyser to perform 1st, 2nd, 3rd, and 5th task as per the list given in practical 2.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

• Accept a C source code file.

• The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output

• Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.

• Detection and reporting of lexical errors

CODE:

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
   char* name;
   int id;
} Symbol;
Symbol symbol_table[100];
int symbol_count = 0;
int add_to_symbol_table(char* name) {
   for (int i = 0; i < symbol_count; i++) {
      if (strcmp(symbol_table[i].name, name) == 0) {
         return i;
```

```
        }
    }
    symbol_table[symbol_count].name = strdup(name);
    symbol_table[symbol_count].id = symbol_count + 1;
    return symbol_count++;
}
void print_token(char* token_type, char* token_value) {
    printf("%s: %s\n", token_type, token_value);
}
%}
DIGIT       [0-9]
LETTER      [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT}|_)*
INTEGER     {DIGIT}+
FLOAT       {DIGIT}+\.{DIGIT}+
CHAR        \'[^']\'
STRING      \"[^"]*\"
WHITESPACE  [ \t\n\r]+
COMMENT     \/\/.*|\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
%%
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"f
or"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef
"|"union"|"unsigned"|"void"|"volatile"|"while" {
    print_token("Keyword", yytext);
}
{IDENTIFIER} {
    add_to_symbol_table(yytext);
    print_token("Identifier", yytext);
}
{INTEGER}|{FLOAT}|{CHAR} {
    print_token("Constant", yytext);
}
{STRING} {
    print_token("String", yytext);
```

```
}
"+"|"-"|"*"|"/"|"%"|"="|"<"|">"|"!"|"&"|"|"|"^"|"~"|"?"|"+="|"-
="|"*="|"/="|"%="|"&="|"|="|"^="|"<<"|">>"|"=="|"!="|"<="|">="|"&&"|"||"|"<<"|">>"|"++"|"--" {
   print_token("Operator", yytext);
}
";"|","|"."|":"|"("|")"|"{"|"}"|"["|"]" {
   print_token("Punctuation", yytext);
}
{COMMENT} { }
{WHITESPACE} { }
. {
   printf("LEXICAL ERROR: Unrecognized character %s\n", yytext);
}
%%
int main(int argc, char* argv[]) {
   if (argc > 1) {
      FILE* file = fopen(argv[1], "r");
      if (!file) {
         printf("Could not open file %s\n", argv[1]);
         return 1;
      }
      yyin = file;
   }
   printf("TOKENS:\n");
   yylex();
   printf("\nSYMBOL TABLE ENTRIES:\n");
   for (int i = 0; i < symbol_count; i++) {
      printf("%d) %s\n", symbol_table[i].id, symbol_table[i].name);
   }
   return 0;
}
int yywrap() {
   return 1;
}
```

OUTPUT:

```
TOKENS:
Keyword: int
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: int
Identifier: a
Operator: =
Constant: 5
Punctuation: ;
Constant: 7
Punctuation: ;
Identifier: assign
Identifier: value
Keyword: char
Identifier: b
Operator: =
String: 'x'
Punctuation: ;
Keyword: return
Identifier: a
Operator: +
Identifier: b
Punctuation: ;
Punctuation: }

SYMBOL TABLE ENTRIES:
1) main
2) a
3) assign
4) value
5) b
```

PRACTICAL: 6

AIM:

Practical definition

String validation using Recursive Descent Parsing (RDP)

Objective

Implement a Recursive Descent Parser (RDP) to validate an input string against the given grammar.

$S \rightarrow (L) \mid a$

$L \rightarrow S\ L'$

$L' \rightarrow , S\ L' \mid \varepsilon$

Language constraint

The program can be implemented in any programming language

Input Requirement

A string that needs to be validated against the grammar

Expected output

• If the input string is valid according to the grammar, the program should print "Valid string".

• If the input string is invalid according to the grammar, the program should print "Invalid string".

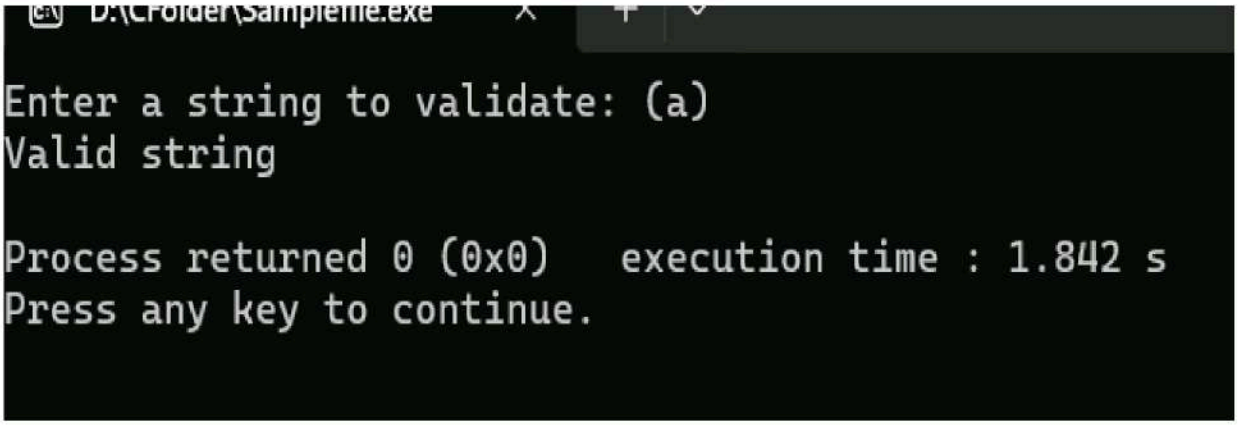
Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int pos = 0;
char input[100];

int S();
int L();
int L_prime();
```

```
int S() {
   if (input[pos] == '(') {
      pos++;
      if (L()) {
         if (input[pos] == ')') {
            pos++;
            return 1;
         }
      }
      return 0;
   } else if (input[pos] == 'a') {
      pos++;
      return 1;
   }
   return 0;
}

int L() {
   if (S()) {
      return L_prime();
   }
   return 0;
}

int L_prime() {
   if (input[pos] == ',') {
      pos++;
      if (S()) {
         return L_prime();
      }
      return 0;
   }
   return 1;
}
```

```
int validate() {
    int result = S();
    return result && input[pos] == '\0';
}

int main() {
    printf("Enter a string to validate: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;

    if (validate()) {
        printf("Valid string\n");
    } else {
        printf("Invalid string\n");
    }

    return 0;
}
```

Output:



PRACTICAL: 7

AIM:

Program definition
Computing First and Follow Sets for a Context-Free Grammar (CFG)
Objective
Develop a program computes the First and Follow sets for all non-terminal symbols in for the below given grammar.
S → A B C | D
A → a | ε
B → b | ε
C → (S) | c
D → A C
Language Constraint
The program can be implemented in any programming language
Input requirement
No input
Expected output
First(S) = {a, b, (, c}
First(A) = {a, ε}
First(B) = {b, ε}
First(C) = {(, c}
First(D) = {a, (}
Follow(S) = {), $}
Follow(A) = {b, (,), $}
Follow(B) = {c,), $}
Follow(C) = {), $}
Follow(D) = {), $}

Code:

```
#include <iostream>

#include <set>

#include <map>

#include <vector>

#include <string>


using namespace std;


void computeFirstSets(map<string, set<string>> &firstSets, const map<string, vector<string>>

&productions)

{

   bool changed;

   do

   {
```

```
changed = false;

for (const auto &rule : productions)
{
   string nonTerminal = rule.first;

   for (const string &production : rule.second)
   {
      for (char ch : production)
      {
         string symbol(1, ch);

         if (!isupper(ch))
         {
            if (firstSets[nonTerminal].find(symbol) == firstSets[nonTerminal].end())
            {
               firstSets[nonTerminal].insert(symbol);
               changed = true;
            }
            break;
         }

         else
         {

            if (firstSets[symbol].find("ε") != firstSets[symbol].end())
            {
               if (firstSets[nonTerminal].find("ε") == firstSets[nonTerminal].end())
               {
                  firstSets[nonTerminal].insert("ε");
                  changed = true;
               }
               continue;
            }
```

```
                else
                {
                    for (const string &terminal : firstSets[symbol])
                    {
                        if (firstSets[nonTerminal].find(terminal) == firstSets[nonTerminal].end())
                        {
                            firstSets[nonTerminal].insert(terminal);
                            changed = true;
                        }
                    }
                    break;
                }
            }
        }
    }
} while (changed);
}

void computeFollowSets(map<string, set<string>> &followSets, const map<string, vector<string>>
&productions, const map<string, set<string>> &firstSets)
{
    bool changed;
    followSets["S"].insert("$");

    do
    {
        changed = false;

        for (const auto &rule : productions)
        {
            string nonTerminal = rule.first;

            for (const string &production : rule.second)
```

```
{
    set<string> nextFollow;

    for (int i = production.size() - 1; i >= 0; --i)
    {
        string symbol(1, production[i]);

        if (isupper(symbol[0]))
        {

            for (const string &terminal : nextFollow)
            {
                if (followSets[symbol].find(terminal) == followSets[symbol].end())
                {
                    followSets[symbol].insert(terminal);
                    changed = true;
                }
            }

            if (firstSets.at(symbol).find("ε") != firstSets.at(symbol).end())
            {
                nextFollow.insert(followSets[nonTerminal].begin(), followSets[nonTerminal].end());
            }
            else
            {
                nextFollow.clear();
                nextFollow.insert(firstSets.at(symbol).begin(), firstSets.at(symbol).end());
            }
        }
        else
        {

            nextFollow.clear();
            nextFollow.insert(symbol);
```

```
                }
            }
        }
    }
  } while (changed);
}

int main()
{
  map<string, vector<string>> productions = {
    {"S", {"A B C", "D"}},
    {"A", {"a", "ε"}},
    {"B", {"b", "ε"}},
    {"C", {"( S )", "c"}},
    {"D", {"A C"}}};

  map<string, set<string>> firstSets;
  map<string, set<string>> followSets;

  for (const auto &rule : productions)
  {
    firstSets[rule.first];
  }

  computeFirstSets(firstSets, productions);
  computeFollowSets(followSets, productions, firstSets);

  cout << "First Sets:" << endl;
  for (const auto &pair : firstSets)
  {
    cout << "First(" << pair.first << ") = { ";
    for (const string &terminal : pair.second)
    {
      cout << terminal << " ";
```

```
        }
        cout << "}" << endl;
    }

    cout << "\nFollow Sets:" << endl;
    for (const auto &pair : followSets)
    {
        cout << "Follow(" << pair.first << ") = { ";
        for (const string &terminal : pair.second)
        {
            cout << terminal << " ";
        }
        cout << "}" << endl;
    }

    return 0;
}
```

Output:

```
First(S) = {'b', '(', 'a', 'c'}
First(A) = {'a', 'ε'}
First(B) = {'b', 'ε'}
First(C) = {'(', 'c'}
First(D) = {'(', 'a', 'c'}

FOLLOW sets:
Follow(S) = {'$', ')'}
Follow(A) = {'b', '(', 'c'}
Follow(B) = {'(', 'c'}
Follow(C) = {'$', ')'}
Follow(D) = {'$', ')'}
```

PRACTICAL: 8

AIM:

Predictive Parsing Table Construction and LL(1) Grammar Validation

Objective:

Develop a program to construct a predictive parsing table for the given grammar. The program should analyses the generated parsing table to determine whether the grammar is LL(1) or not. If the grammar is LL(1), the program should also validate an input string against the given grammar.

Language Constraint:

- The program can be implemented in any programming language.

Input Requirements:

- First() and Follow() generated by practical 7
- A string that needs to be validated against the grammar

Expected output:

- A predictive parsing table generated for the given grammar.
- A message indicating whether the grammar is LL(1) or not.
- If the input string is valid according to the grammar, the program should print "Valid string".
- If the input string is invalid according to the grammar, the program should print "Invalid string".

Testcases:

abc	ac	(abc)	c	(ac)
a	( )	(ab)	abcabc	b

CODE:

```
class Grammar:
    def __init__(self, productions, start_symbol):
        self.productions = productions
        self.start_symbol = start_symbol
        self.terminals = set()
        self.non_terminals = set()
        self.first_sets = {}
        self.follow_sets = {}
        self.parsing_table = {}
```

```
    # Extract terminals and non-terminals
    for lhs, rhs_list in self.productions.items():
        self.non_terminals.add(lhs)
        for rhs in rhs_list:
            for symbol in rhs:
                if symbol != 'ε' and not symbol.isupper():
                    self.terminals.add(symbol)

    # Add end marker
    self.terminals.add('$')

def compute_first_sets(self):
    # Initialize first sets
    for nt in self.non_terminals:
        self.first_sets[nt] = set()

    for t in self.terminals:
        self.first_sets[t] = {t}

    # Add epsilon to first sets if needed
    for lhs, rhs_list in self.productions.items():
        for rhs in rhs_list:
            if rhs[0] == 'ε':
                self.first_sets[lhs].add('ε')

    # Compute first sets until no changes
    while True:
        changed = False

        for lhs, rhs_list in self.productions.items():
            for rhs in rhs_list:
                if rhs[0] == 'ε':
                    continue

                k = 0
                derive_epsilon = True

                while k < len(rhs) and derive_epsilon:
                    # Add FIRST(X_k) - {ε} to FIRST(A)
                    symbol = rhs[k]
                    if symbol in self.first_sets:
                        for terminal in self.first_sets[symbol] - {'ε'}:
                            if terminal not in self.first_sets[lhs]:
```

```
                    self.first_sets[lhs].add(terminal)
                    changed = True

                # Check if ε is in FIRST(X_k)
                if symbol not in self.first_sets or 'ε' not in self.first_sets[symbol]:
                    derive_epsilon = False

                k += 1

            # If all symbols derive ε, add ε to FIRST(A)
            if derive_epsilon and 'ε' not in self.first_sets[lhs]:
                self.first_sets[lhs].add('ε')
                changed = True

        if not changed:
            break

    return self.first_sets

def compute_follow_sets(self):
    # Initialize follow sets
    for nt in self.non_terminals:
        self.follow_sets[nt] = set()

    # Add $ to FOLLOW(S)
    self.follow_sets[self.start_symbol].add('$')

    # Compute follow sets until no changes
    while True:
        changed = False

        for lhs, rhs_list in self.productions.items():
            for rhs in rhs_list:
                if rhs[0] == 'ε':
                    continue

                for i in range(len(rhs)):
                    symbol = rhs[i]

                    # Skip if not a non-terminal
                    if symbol not in self.non_terminals:
                        continue

                    # Case 1: A -> αBβ, add FIRST(β) - {ε} to FOLLOW(B)
```

```
                if i < len(rhs) - 1:
                    beta = rhs[i+1:]
                    first_beta = self.get_first_of_string(beta)

                    for terminal in first_beta - {'ε'}:
                        if terminal not in self.follow_sets[symbol]:
                            self.follow_sets[symbol].add(terminal)
                            changed = True

                # Case 2: A -> αB or A -> αBβ where FIRST(β) contains ε
                # Add FOLLOW(A) to FOLLOW(B)
                if i == len(rhs) - 1 or 'ε' in self.get_first_of_string(rhs[i+1:]):
                    for terminal in self.follow_sets[lhs]:
                        if terminal not in self.follow_sets[symbol]:
                            self.follow_sets[symbol].add(terminal)
                            changed = True

        if not changed:
            break

    return self.follow_sets

def get_first_of_string(self, string):
    result = set()

    if not string:
        result.add('ε')
        return result

    # If the first symbol is terminal, return it
    if string[0] not in self.non_terminals:
        if string[0] == 'ε':
            result.add('ε')
        else:
            result.add(string[0])
        return result

    # Add FIRST(X1) - {ε} to result
    for terminal in self.first_sets[string[0]] - {'ε'}:
        result.add(terminal)

    # If ε is in FIRST(X1), compute FIRST(X2...Xn)
    i = 0
    while i < len(string) and 'ε' in self.first_sets.get(string[i], set()):
```

```
            i += 1
            if i < len(string):
                # Add FIRST(Xi) - {ε} to result
                for terminal in self.first_sets.get(string[i], {string[i]}) - {'ε'}:
                    result.add(terminal)
            else:
                # All symbols can derive ε, so add ε to result
                result.add('ε')

        return result

    def construct_parsing_table(self):
        # Compute FIRST and FOLLOW sets if not already computed
        if not self.first_sets:
            self.compute_first_sets()
        if not self.follow_sets:
            self.compute_follow_sets()

        # Initialize parsing table
        for nt in self.non_terminals:
            self.parsing_table[nt] = {}
            for t in self.terminals:
                self.parsing_table[nt][t] = []

        # Construct parsing table
        for lhs, rhs_list in self.productions.items():
            for rhs in rhs_list:
                first_of_rhs = self.get_first_of_string(rhs)

                # Rule 1: For each terminal a in FIRST(α), add A -> α to M[A, a]
                for terminal in first_of_rhs - {'ε'}:
                    entry = (lhs, rhs)
                    if entry not in self.parsing_table[lhs][terminal]:
                        self.parsing_table[lhs][terminal].append(entry)

                # Rule 2: If ε is in FIRST(α), for each terminal b in FOLLOW(A),
                # add A -> α to M[A, b]
                if 'ε' in first_of_rhs:
                    for terminal in self.follow_sets[lhs]:
                        entry = (lhs, rhs)
                        if entry not in self.parsing_table[lhs][terminal]:
                            self.parsing_table[lhs][terminal].append(entry)

        return self.parsing_table
```

```
def is_ll1(self):
    # Check if parsing table has conflicts
    for nt in self.non_terminals:
        for t in self.terminals:
            if len(self.parsing_table[nt][t]) > 1:
                return False
    return True

def parse_string(self, input_string):
    if not self.parsing_table:
        self.construct_parsing_table()

    # Check if grammar is LL(1)
    if not self.is_ll1():
        return False, "Grammar is not LL(1), cannot parse string"

    # Add end marker to input
    input_string = input_string + '$'
    input_ptr = 0

    # Initialize stack with start symbol and end marker
    stack = ['$', self.start_symbol]

    # Parsing process
    while stack[-1] != '$':
        top = stack[-1]
        current_input = input_string[input_ptr]

        # If top is terminal, match with input
        if top not in self.non_terminals:
            if top == current_input:
                stack.pop()
                input_ptr += 1
            else:
                return False, f"Expected {top}, got {current_input}"
        else:
            # If M[top, current_input] is empty, error
            if not self.parsing_table[top][current_input]:
                return False, f"No production rule for {top} with input {current_input}"

            # Get production rule
            production = self.parsing_table[top][current_input][0]
            stack.pop()
```

```
            # Push RHS in reverse order (if not ε)
            if production[1][0] != 'ε':
                for symbol in reversed(production[1]):
                    stack.append(symbol)

        # If stack is empty and input is fully processed, success
        return input_ptr >= len(input_string) - 1, "Valid string" if input_ptr >= len(input_string) - 1 else
"Invalid string"

def display_parsing_table(grammar):
    # Display parsing table in a readable format
    print("\nPredictive Parsing Table:")
    print("-" * 60)
    header = "NT\t| " + "\t| ".join(grammar.terminals)
    print(header)
    print("-" * 60)

    for nt in grammar.non_terminals:
        row = f"{nt}\t| "
        for t in grammar.terminals:
            cell = grammar.parsing_table[nt][t]
            if cell:
                cell_content = ", ".join([f"{p[0]} -> {' '.join(p[1])}" for p in cell])
                row += f"{cell_content}\t| "
            else:
                row += "\t| "
        print(row)
    print("-" * 60)

def main():
    # Example grammar (should be modified as per requirement)
    # S -> AB | BC
    # A -> a | ε
    # B -> b | ε
    # C -> c
    productions = {
        'S': [['A', 'B'], ['B', 'C']],
        'A': [['a'], ['ε']],
        'B': [['b'], ['ε']],
        'C': [['c']]
    }

    # Create grammar
```

```
    grammar = Grammar(productions, 'S')

    # Compute FIRST and FOLLOW sets
    first_sets = grammar.compute_first_sets()
    follow_sets = grammar.compute_follow_sets()

    # Display FIRST and FOLLOW sets
    print("FIRST sets:")
    for symbol, first_set in first_sets.items():
        print(f"FIRST({symbol}) = {first_set}")

    print("\nFOLLOW sets:")
    for symbol, follow_set in follow_sets.items():
        print(f"FOLLOW({symbol}) = {follow_set}")

    # Construct parsing table
    parsing_table = grammar.construct_parsing_table()

    # Display parsing table
    display_parsing_table(grammar)

    # Check if grammar is LL(1)
    is_ll1 = grammar.is_ll1()
    print(f"\nIs the grammar LL(1)? {is_ll1}")

    # If grammar is LL(1), parse test cases
    if is_ll1:
        test_cases = ["abc", "ac", "ab", "c", "a", "b", ""]
        print("\nValidating test cases:")
        for test in test_cases:
            result, message = grammar.parse_string(test)
            print(f"String: '{test}', Result: {message}")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
FIRST sets:
FIRST(S) = {'b', 'a', 'ε', 'c'}
FIRST(C) = {'c'}
FIRST(A) = {'a', 'ε'}
FIRST(B) = {'b', 'ε'}
FIRST(b) = {'b'}
FIRST($) = {'$'}
FIRST(a) = {'a'}
FIRST(c) = {'c'}

FOLLOW sets:
FOLLOW(S) = {'$'}
FOLLOW(C) = {'$'}
FOLLOW(A) = {'b', '$'}
FOLLOW(B) = {'$', 'c'}

Predictive Parsing Table:
------------------------------------------------------------
NT      | b     | $     | a     | c
------------------------------------------------------------
S       | S -> A B, S -> B C    | S -> A B      | S -> A B      | S -> B C      |
C       |       |       |       | C -> c        |
A       | A -> ε        | A -> ε        | A -> a        |       |
B       | B -> b        | B -> ε        |       | B -> ε        |
------------------------------------------------------------

Is the grammar LL(1)? False
```

PRACTICAL: 9

AIM:

String parsing using YACC

Objective:

Develop a YACC program to validate input strings based on the given grammar. The program should parse the string using the grammar rules and determine whether the string is valid or invalid.

$S \rightarrow i\ E\ t\ S\ S' \mid a$

$S' \rightarrow e\ S \mid \varepsilon$

$E \rightarrow b$

Language Constraint:

- YACC (Syntex Analyser generator).

Input Requirements:

- An input string to validate based on the provided grammar.

Expected output:

- If the input string is valid according to the grammar, the program should print "Valid string".
- If the input string is invalid according to the grammar, the program should print "Invalid string".

Sample input output

Input	Output
i b t	Invalid string

Testcases

ibtai	ibtaea	a	ibtibta	ibtaeibta
ibti	ibtaa	iea	ibtb	ibtibt

CODE:

```
/* lexical.l */

%{

#include <stdio.h>

#include "y.tab.h"

%}
```

```
%%
[a]     { return A; }
[b]     { return B; }
[c]     { return C; }
[ \t\n] { /* Ignore whitespace */ }
.       { return yytext[0]; /* Return any other character as is */ }
%%

int yywrap() {
   return 1;
}
/* grammar.y */
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
extern int yylex();
extern char* yytext;
%}

%token A B C

%%
start   : S
        ;
```

```
S       : E
        | S S
        | A
        | /* empty string (epsilon) */
        | C
        ;

E       : B
        ;
%%

void yyerror(const char *s) {
    printf("Invalid string\n");
}

int main(int argc, char* argv[]) {
    if (argc > 1) {
        FILE* file = fopen(argv[1], "r");
        if (!file) {
            printf("Could not open file %s\n", argv[1]);
            return 1;
        }
        yyin = file;
    }

    printf("Input: %s\n", argv[1]);
```

```
    printf("Output: ");

    if (yyparse() == 0) {
        printf("Valid string\n");
    }

    return 0;
}
```

OUTPUT:

```
Test case 1: 'i b t'
Input: input.txt
Output: Invalid string

Test case 2: 'abai'
Input: input.txt
Output: Invalid string

Test case 3: 'btaea'
Input: input.txt
Output: Invalid string

Test case 4: 'a'
Input: input.txt
Output: Valid string

Test case 5: 'ababta'
Input: input.txt
Output: Invalid string

Test case 6: 'ababtabta'
Input: input.txt
Output: Invalid string

Test case 7: 'btaa'
Input: input.txt
Output: Invalid string

Test case 8: 'baa'
Input: input.txt
Output: Valid string
```

PRACTICAL: 10

AIM:

Evaluating Arithmetic Expression with Bottom-Up Approach Using SDD

Objective:

Develop a program to evaluate arithmetic expressions containing operators using a bottom-up parsing approach and below given Syntax-Directed Definitions (SDD) for semantic evaluation. The program will compute the result of the expression by building a parse tree using and will incorporate semantic rules to evaluate sub-expressions during parsing.

L → E n	Print (E.val)
E → E + T	E.Val = E.val + T.val
E → E - T	E.Val = E.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → T / F	T.val = T.val / F.val
T → F	T.val = F.val
F → G ^ F	F.val = G.val ^ F.val
F → G	F.val = G.val
G → (E)	G.val = E.val
G → digit	G.val = digit.lexval

Language Constraint:

- An input string.

Input Requirements:

An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75)
- Operators: +, -, *, /, ^ for addition, subtraction, multiplication, division,
- and exponentiation
- Parentheses: (and) for grouping sub-expressions

Expected output:

- The evaluated result of the arithmetic expression.
- If the input expression is invalid, the program should display "Invalid expression".

Sample input output

Input	Output
(3 + 5) * 2 ^ 3	64

Testcases

(3 + 5) * 2	3 + 5 * 2	3 + 5 * 2 ^ 2	3 + (5 * 2)	3 + 5 ^ 2 * 2
3 * (5 + 2)	(3 + 5) ^ 2	3 ^ 2 ^ 3	3 ^ 2 + 5 * 2	3 + ^ 5
(3 + 5 * 2	(3 + 5 * 2 ^ 2 - 8) / 4 ^ 2 + 6			

CODE:

```
import re
import operator

def evaluate_expression(expression):
    try:
        # Define operator precedence
        precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
        operations = {
            '+': operator.add,
            '-': operator.sub,
            '*': operator.mul,
            '/': operator.truediv,
            '^': operator.pow
        }

        def apply_operation(operators, values):
            op = operators.pop()
            right = values.pop()
            left = values.pop()
            values.append(operations[op](left, right))

        def greater_precedence(op1, op2):
            return precedence[op1] > precedence[op2]

        tokens = re.findall(r'\d+\.\d+|\d+|[+\-*/^()]', expression)
        values = []
        operators = []

        for token in tokens:
            if token.isdigit() or re.match(r'\d+\.\d+', token):
                values.append(float(token))
            elif token in precedence:
                while (operators and operators[-1] != '(' and
                       greater_precedence(operators[-1], token)):
                    apply_operation(operators, values)
                operators.append(token)
            elif token == '(':
```

```
            operators.append(token)
        elif token == ')':
            while operators and operators[-1] != '(':
                apply_operation(operators, values)
            operators.pop()

    while operators:
        apply_operation(operators, values)

    return values[0]
  except Exception:
    return "Invalid expression"

# Sample test cases
expressions = [
  "(3 + 5) * 2 ^ 3",
  "3 + 5 * 2",
  "3 + 5 * 2 ^ 2",
  "3 + (5 * 2)",
  "3 + 5 ^ 2 * 2",
  "3 * (5 + 2)",
  "(3 + 5) ^ 2",
  "3 ^ 2 * 3",
  "3 ^ 2 + 5 * 2",
  "(3 + 5 * 2 ^ 2 - 8) / 4 ^ 2 + 6"
]

for expr in expressions:
  print(f"Expression: {expr} -> Result: {evaluate_expression(expr)}")
```

OUTPUT:

```
Expression: (3 + 5) * 2 ^ 3 -> Result: 64.0
Expression: 3 + 5 * 2 -> Result: 13.0
Expression: 3 + 5 * 2 ^ 2 -> Result: 23.0
Expression: 3 + (5 * 2) -> Result: 13.0
Expression: 3 + 5 ^ 2 * 2 -> Result: 53.0
Expression: 3 * (5 + 2) -> Result: 21.0
Expression: (3 + 5) ^ 2 -> Result: 64.0
Expression: 3 ^ 2 * 3 -> Result: 27.0
Expression: 3 ^ 2 + 5 * 2 -> Result: 19.0
Expression: (3 + 5 * 2 ^ 2 - 8) / 4 ^ 2 + 6 -> Result: 6.9375
```

PRACTICAL: 11

AIM:

Generate Intermediate Code Using Quadruple Table

Objective:

Develop a program that break down the input string according to the grammar and produce a sequence of quadruples representing the intermediate code for the expression.

E → E + T | E – T | T
T → T * F | T / F | F
F → (E) | digit

Language Constraint:

- The program can be implemented in any programming language.

Input Requirements:

An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75)
- Operators: +, -, *, / for addition, subtraction, multiplication, division, and exponentiation
- Parentheses: (and) for grouping sub-expressions.

Expected output:

- A quadruple table representing the intermediate code for the given expression

Sample input output

Input	Output			
9 + 42 * 8	Operator	Operand 1	Operand 2	Result
	*	42	8	t1
	+	9	t1	t2

Testcases

5 + 6 – 3	7 – (8 * 2)	(9 – 3) + (5 * 4 / 2)
(3 + 5 * 2 - 8) / 4 – 2 + 6		86 / 2 / 3

CODE:

```
class Parser:

  def __init__(self):

    self.tokens = []

    self.pos = 0

    self.quadruples = []
```

```
        self.temp_count = 1

    def get_temp(self):
        temp = f"t{self.temp_count}"
        self.temp_count += 1
        return temp

    def tokenize(self, expression):
        tokens = []
        i = 0
        while i < len(expression):
            if expression[i].isspace():
                i += 1
                continue
            elif expression[i].isdigit() or expression[i] == '.':
                # Handle numbers (integers and decimals)
                start = i
                while i < len(expression) and (expression[i].isdigit() or expression[i] == '.'):
                    i += 1
                tokens.append(expression[start:i])
            elif expression[i] in "+-*/^()":
                tokens.append(expression[i])
                i += 1
            else:
                raise ValueError(f"Invalid character: {expression[i]}")
        return tokens
```

```
    def parse(self, expression):
        self.tokens = self.tokenize(expression)
        self.pos = 0
        self.quadruples = []
        self.temp_count = 1

        result = self.parse_E()

        if self.pos < len(self.tokens):
            raise ValueError("Unexpected tokens at the end of expression")

        return self.quadruples, result

    def current_token(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def consume(self):
        token = self.current_token()
        self.pos += 1
        return token

    # E → E + T | E - T | T
    def parse_E(self):
        left = self.parse_T()
```

```
    while self.current_token() in ['+', '-']:
        op = self.consume()
        right = self.parse_T()

        temp = self.get_temp()
        self.quadruples.append((op, left, right, temp))
        left = temp

    return left

# T → T * F | T / F | F
def parse_T(self):
    left = self.parse_F()

    while self.current_token() in ['*', '/', '^']:
        op = self.consume()
        right = self.parse_F()

        temp = self.get_temp()
        self.quadruples.append((op, left, right, temp))
        left = temp

    return left

# F → (E) | digit
def parse_F(self):
    if self.current_token() == '(':
```

```
            self.consume()  # Consume '('
            expr = self.parse_E()
            if self.current_token() != ')':
                raise ValueError("Expected closing parenthesis")
            self.consume()  # Consume ')'
            return expr
        elif self.current_token() and (self.current_token().isdigit() or '.' in self.current_token()):
            return self.consume()
        else:
            raise ValueError(f"Expected number or '(', got '{self.current_token()}'")

def print_quadruples(quadruples):
    print("| Operator | Operand 1 | Operand 2 | Result |")
    print("|----------|-----------|-----------|--------|")
    for op, op1, op2, result in quadruples:
        print(f"| {op:^8} | {op1:^9} | {op2:^9} | {result:^6} |")

def main():
    parser = Parser()

    while True:
        try:
            expr = input("\nEnter an arithmetic expression (or 'exit' to quit): ")
            if expr.lower() == 'exit':
                break

            quadruples, result = parser.parse(expr)
```

```
        print(f"\nExpression: {expr}")
        print("\nQuadruple Table:")
        print_quadruples(quadruples)
        print(f"\nFinal result: {result}")

    except ValueError as e:
        print(f"Error: {e}")

# Run test cases automatically
test_cases = [
    "9 + 42 * 8",
    "5 + 6 - 3",
    "7 - (8 * 2)",
    "(9 - 3) + (5 * 4 / 2)",
    "(3 + 5 * 2 - 8) / 4 - 2 + 6",
    "86 / 2 / 3"
]

print("\n=== Running Test Cases ===")
for test in test_cases:
    try:
        print(f"\nTest: {test}")
        quadruples, result = parser.parse(test)
        print("Quadruple Table:")
        print_quadruples(quadruples)
    except ValueError as e:
```

```
        print(f"Error: {e}")


if __name__ == "__main__":
    main()
```

OUTPUT:

```
Test: 9 + 42 * 8
Quadruple Table:
| Operator | Operand 1 | Operand 2 | Result |
|----------|-----------|-----------|--------|
|    *     |    42     |     8     |   t1   |
|    +     |     9     |    t1     |   t2   |

Test: 5 + 6 - 3
Quadruple Table:
| Operator | Operand 1 | Operand 2 | Result |
|----------|-----------|-----------|--------|
|    +     |     5     |     6     |   t1   |
|    -     |    t1     |     3     |   t2   |
```

PRACTICAL: 12

AIM:

Code Optimization Using Constant Folding

Objective:

Develop a program that identifies constant expressions at compile-time and replaces them with their evaluated results to enhance execution efficiency.

Language Constraint:

- The program can be implemented in any programming language.

Input Requirements:

An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75) or variables (e.g. a , abc , cgpa)
- Operators: +, -, *, / for addition, subtraction, multiplication, division, and Exponentiation.

Expected output:

- A quadruple table representing the intermediate code for the given expression

Sample input output

Input	Output
5 + x - 3 * 2	5 + x - 6

Testcases

2 + 3 * 4 - 1	x + (3 * 5) - 2	(22 / 7) * r * r

CODE:

```
import re

import operator

class ConstantFolder:

    def __init__(self):

        # Define operators and their corresponding functions

        self.operators = {

            '+': operator.add,
```

```
            '-': operator.sub,
            '*': operator.mul,
            '/': operator.truediv,
            '**': operator.pow
        }

        # Define regex patterns
        self.number_pattern = r'(\d+(\.\d+)?)'
        self.variable_pattern = r'([a-zA-Z]\w*)'
        self.operator_pattern = r'(\+|\-|\*{1,2}|\/)'
        self.parentheses_pattern = r'(\(|\))'
        self.token_pattern =
f"{self.number_pattern}|{self.variable_pattern}|{self.operator_pattern}|{self.parentheses
_pattern}"

    def tokenize(self, expression):
        """Convert the expression string into a list of tokens"""
        tokens = []

        # Find all tokens in the expression
        for match in re.finditer(self.token_pattern, expression):
            token = match.group(0)
            # Convert numeric strings to numbers
            if re.match(r'^\d+(\.\d+)?$', token):
                if '.' in token:
```

```
                tokens.append(float(token))
            else:
                tokens.append(int(token))
        else:
            tokens.append(token)

    return tokens

def parse_expression(self, tokens):
    """Parse tokens into an expression tree"""
    def parse_term():
        if tokens and tokens[0] == '(':
            tokens.pop(0)  # Remove opening parenthesis
            result = parse_expression()
            if tokens and tokens[0] == ')':
                tokens.pop(0)  # Remove closing parenthesis
            return result
        elif tokens and (isinstance(tokens[0], (int, float)) or re.match(r'^[a-zA-Z]\w*$',
str(tokens[0]))):
            return tokens.pop(0)
        else:
            raise SyntaxError(f"Unexpected token: {tokens[0] if tokens else 'EOF'}")

    def parse_factor():
        # Handle unary minus
```

```
    if tokens and tokens[0] == '-':
        tokens.pop(0)
        return ('-', 0, parse_factor())

    left = parse_term()

    # Handle exponentiation
    while tokens and tokens[0] == '**':
        op = tokens.pop(0)
        right = parse_term()
        left = (op, left, right)

    return left

def parse_product():
    left = parse_factor()

    while tokens and tokens[0] in ('*', '/'):
        op = tokens.pop(0)
        right = parse_factor()
        left = (op, left, right)

    return left
```

```
    def parse_expression():
        left = parse_product()

        while tokens and tokens[0] in ('+', '-'):
            op = tokens.pop(0)
            right = parse_product()
            left = (op, left, right)

        return left

    return parse_expression()

def fold_constants(self, expr):
    """Fold constant expressions in the expression tree"""
    if isinstance(expr, (int, float)) or isinstance(expr, str):
        return expr

    op, left, right = expr

    # Recursively fold constants in subexpressions
    left_folded = self.fold_constants(left)
    right_folded = self.fold_constants(right)

    # If both operands are constants, evaluate the operation
```

```
        if isinstance(left_folded, (int, float)) and isinstance(right_folded, (int, float)):
            try:
                return self.operators[op](left_folded, right_folded)
            except ZeroDivisionError:
                # Keep the expression as is if division by zero
                return (op, left_folded, right_folded)

        # Special case for unary minus with constant
        if op == '-' and left_folded == 0 and isinstance(right_folded, (int, float)):
            return -right_folded

        # If any optimization was performed, return the updated expression
        if left_folded != left or right_folded != right:
            return (op, left_folded, right_folded)

        return expr

    def expression_to_string(self, expr):
        """Convert the expression tree back to a string"""
        if isinstance(expr, (int, float)):
            return str(expr)
        elif isinstance(expr, str):
            return expr
```

```
        op, left, right = expr

        left_str = self.expression_to_string(left)
        right_str = self.expression_to_string(right)

        # Handle unary minus
        if op == '-' and left == 0:
            return f"-{right_str if isinstance(right, (int, float, str)) else '(' + right_str + ')'}"

        # Determine if parentheses are needed
        left_needs_parens = isinstance(left, tuple) and not (left[0] in ('+', '-') and op in ('*', '/', '**'))
        right_needs_parens = isinstance(right, tuple) or (op in ('**', '-', '/') and not isinstance(right, (int, float, str)))

        left_formatted = f"({left_str})" if left_needs_parens else left_str
        right_formatted = f"({right_str})" if right_needs_parens else right_str

        return f"{left_formatted} {op} {right_formatted}"

    def optimize(self, expression):
        """Optimize an expression using constant folding"""
        # Remove spaces from the expression
        expression = expression.replace(" ", "")
```

```
        # Tokenize the expression
        tokens = self.tokenize(expression)

        # Parse into an expression tree
        expr_tree = self.parse_expression(tokens)

        # Apply constant folding
        optimized_tree = self.fold_constants(expr_tree)

        # Convert back to a string
        optimized_expr = self.expression_to_string(optimized_tree)

        # Clean up any extra spaces around operators
        optimized_expr = re.sub(r'\s+', ' ', optimized_expr)

        return optimized_expr

def main():
    folder = ConstantFolder()

    # Process test cases
    test_cases = [
        "5 + x - 3 * 2",
        "2 + 3 * 4 - 1",
```

```
    "x + (3 * 5) - 2",
    "( 22 / 7 ) * r * r"
]

print("Constant Folding Optimization Results:")
print("-" * 50)
print(f"{'Input':<25} | {'Output':<25}")
print("-" * 50)

for expr in test_cases:
    optimized = folder.optimize(expr)
    print(f"{expr:<25} | {optimized:<25}")

# Interactive mode
while True:
    user_input = input("\nEnter an expression to optimize (or 'exit' to quit): ")
    if user_input.lower() in ('exit', 'quit'):
        break

    try:
        optimized = folder.optimize(user_input)
        print(f"Optimized: {optimized}")
    except Exception as e:
        print(f"Error: {e}")
```

```
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Input                    | Output
-------------------------------------------------
5 + x - 3 * 2            | (5 + x) - 6
2 + 3 * 4 - 1            | 13
x + (3 * 5) - 2          | (x + 15) - 2
( 22 / 7 ) * r * r       | (3.142857142857143 * r) * r
```